SAND2015-6867 TR

Exceptional service in the national interest

Dakota Software Training

Interfacing to a Simulation

http://dakota.sandia.gov

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Module Learning Goals

Understand:

- … the mechanics of how Dakota communicates with and runs a simulation
- … the requirements this places on the user and interface
- … some basic strategies for developing a simulation interface
- … some of the convenience features Dakota provides for managing simulation runs

Module Outline

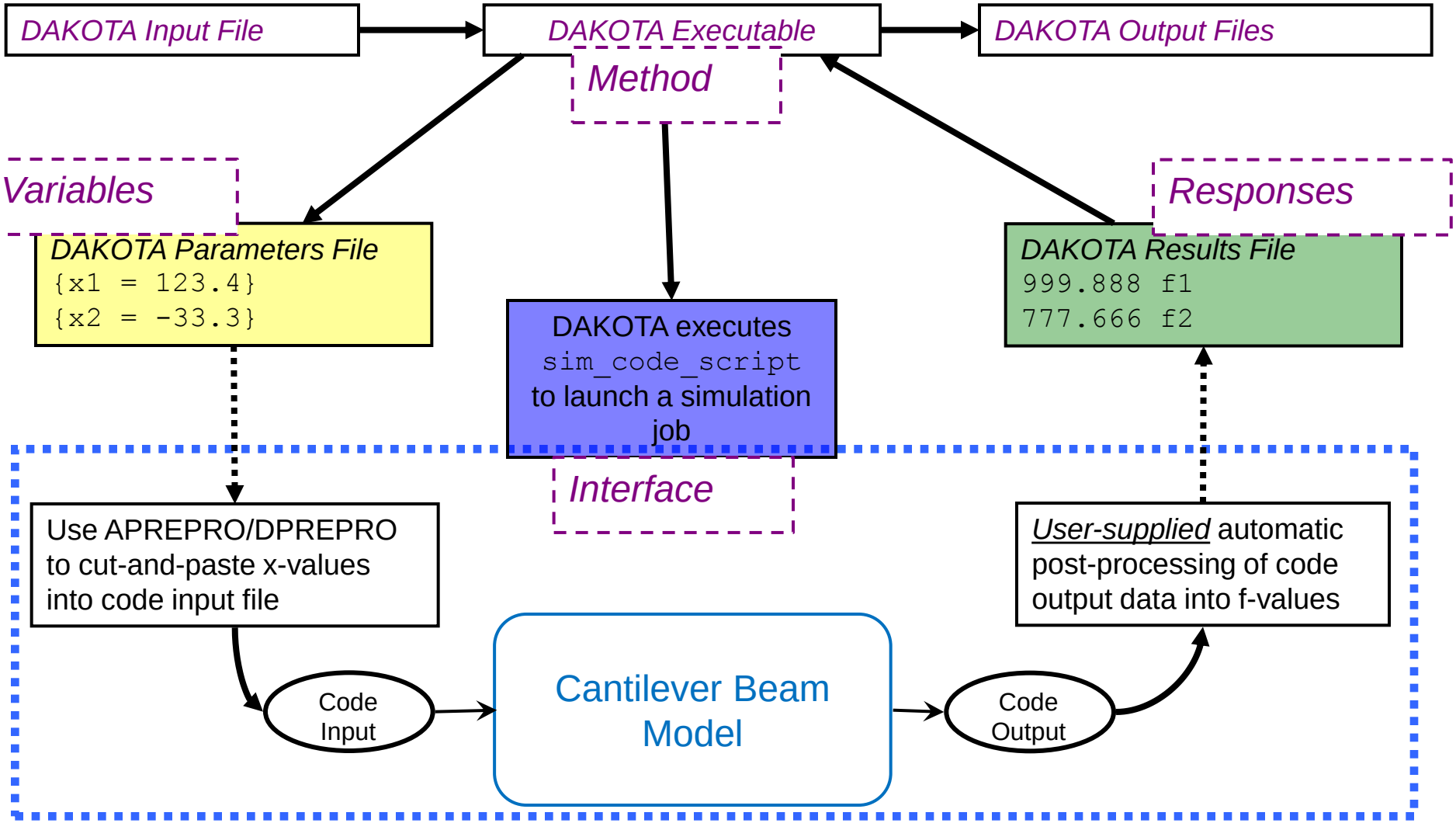
- What happens during a Dakota evaluation
- Considerations for creating a parameterized workflow
- Pre-processing
- Post-processing
- Demonstration
- Exercise
- Dakota input interface specification

Sandia National Laboratories

Interfacing

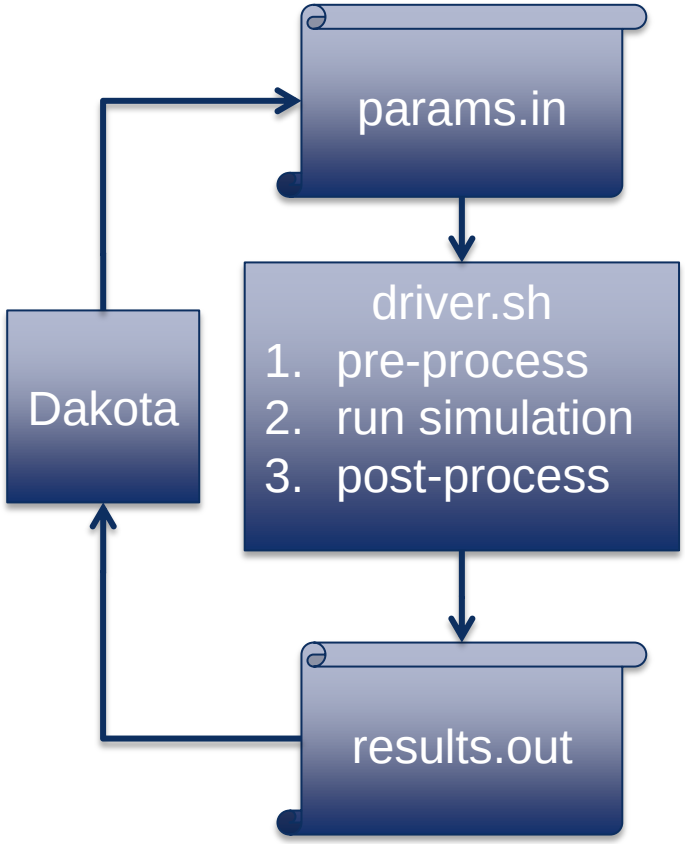
WHAT HAPPENS DURING A DAKOTA EVALUATION

Dakota to Simulation Workflow



A Typical Dakota Evaluation

1. Dakota writes a parameters file that contains one value for each variable
2. Dakota invokes the user's interface, passing to it the filesystem path/names of the parameters and results files as command line arguments
3. The user's interface driver performs three tasks:
   1. **Pre-processing:** Create simulation input using values from the Dakota parameters file
   2. **Run:** Run the simulation based on the input
   3. **Post-processing:** Extract scalar quantities of interest (responses) from simulation output and write them to the named Dakota results file
4. The user's interface exits
5. Dakota opens and reads the results file



Dakota Input: Simulation Contract

- The **variables** block dictates what Dakota will place in the parameters file
- The **interface** block indicates what driver will run to perform the mapping
- The **response** block dictates what Dakota expects back

```
variables
  active all
  continuous_design = 3
    initial_point  2*1.0  10.0
    descriptors    "w"     "t"    "L"
  continuous_state = 4
    initial_state   500. 29.E+6   5.  10.
    descriptors      'p'   'E'    'X'  'Y'

interface
  analysis_driver = 'driver.sh'

responses
  response_functions = 3
  descriptors = 'mass'
                'stress'
                'displacement'
  no_gradients no_hessians
```

Variables with descriptors:
w, *t*, *L*, *p*, *E*, *X*, *and Y*

Responses with descriptors:
mass, *stress*, *displacement*

Interfacing Preparedness

- "Parameterized" simulations/workflows
  - Must know what your parameters are
  - Tough if parameters are hard-coded
- Can my analysis be automated/scripted?
  - Does your workflow depend on tools that are challenging to automate on your target platform?
  - Is the simulation robust to parameter variations?
- Quantities of Interest (QoIs)
  - As with parameters, must know what the responses are
  - Can you extract them automatically?
  - If your QoI is poorly behaved (nonsmooth, noisy), is there another you could choose?

Interfacing

PRE-PROCESSING

Pre-processing

What it is: Converting a Dakota parameters file into usable input for your simulation

Example tasks:

- Substituting parameter values into a text-based input deck ("configuring" an input file)
- Passing parameter values to a simulation as command line arguments, e.g., `how2getrich.exe --param1=42 --param2=-1.4`
- Running a script or program to generate more complex input based on parameter values, e.g. parameterized mesh or geometry in a finite element analysis

Parameters File Format

Dakota uses a parameters file to inform your code of parameter values and to request responses. (Secs. 9.6 & 9.7 of the User's Manual for more Info)

```
                        6 variables
2.50000000000000e+000 w
2.50000000000000e+000 t
4.00000000000000e+004 R
2.90000000000000e+007 E
5.00000000000000e+002 X
1.00000000000000e+003 Y
                        3 functions
                        1 ASV_1:mass
                        1 ASV_2:stress
                        1 ASV_3:displacement
                        6 derivative_variables
                        1 DVV_1:w
                        2 DVV_2:t
                        3 DVV_3:R
                        4 DVV_4:E
                        5 DVV_5:X
                        6 DVV_6:Y
                        0 analysis_components
                        1 eval_id
```

Parameter Values

Requested Responses

Parameter Substitution with dprepro

dprepro (**D**akota **Pre**-**Pro**cessor) is a Dakota-team developed command line tool to simplify parameter substitution.

1. Create a template input file for your simulation with parameter values replaced by substitution tokens.

   Tokens look like: `{dakota_descriptor}`

2. Run dprepro. It replaces tokens with corresponding variable values. Syntax:

   ```
   dprepro <parameters file> <template> <input file>
   ```

Parameter Substitution with dprepro

```
variables
 continuous design = 2
   descriptors “dak_x1" “dak_x2"
```

← Dakota input file

input.template:
User-created template →

```
x1 = {dak_x1}
x2 = {dak_x2}
```

params.in:
Dakota parameters file →

```
                                2 variables
   2.5000000000000000e+000 dak_x1
   4.5000000000000000e+000 dak_x2
```

```
dprepro params.in input.template myinput.in
```

```
x1 = 2.5
x2 = 4.5
```

← myinput.in:
Simulation Input file, written by dprepro

Additional dprepro Features

- Token delimiters (left and right braces) can be changed to other characters or strings:

```
dprepro --left-delimiter='[[' --right-delimiter=']]' ...
```

- Simple arithmetic statements within tokens will be evaluated.
  - `{10.0**dakota_descriptor}`
  - `{dakota_descriptor_1 * dakota_descriptor_2}`

- Default parameters are also supported:
  - `{dakota_descriptor = 3.0}` → replaced by the value of `dakota_descriptor` if present in the parameters file, but 3.0 otherwise.

Interfacing

POST-PROCESSING

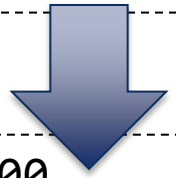
Post-processing

What it is: Extracting quantities of interest from simulation output and placing them in a property formatted and named Dakota results file

- Extracting QoIs can be challenging—there are no shortcuts like dprepro!
- If the QoI appears in console or text file output, time-honored *nix text processing commands can be used to extract it:
  - grep, cut, awk, head, tail, tr, bc, etc.
  - See http://www.tldp.org/LDP/abs/html/textproc.html for a nice list with descriptions, instructions, and examples.
- It may be necessary to calculate a QoI from your output, like the max, min, or integral of a trend

Results File Format

```
responses
  response_functions = 3
  descriptors = ‘mass'
                'stress'
                'displacement'
  no_gradients no_hessians
```



```
6.2500000000e+00     mass
1.7600000000e+04     stress
1.6943165672e+00     displacement
```

- One response per line
- Value (format unimportant) followed by (optional) descriptor
- Descriptors are ignored; order is what matters!
- No spaces in descriptors

(See Chapters 10 & 11 of the User’s Manual for more information.)

Discussion: Post-processing

- Share with your neighbor a quantity of interest you wish to study with Dakota.
- In what kind of output file does it appear?
- Is it easy to extract for return to Dakota? Why or why not?

Discussion: Post-processing

- Share with your neighbor a quantity of interest you wish to study with Dakota.
- In what kind of output file does it appear?
- Is it easy to extract for return to Dakota? Why or why not?

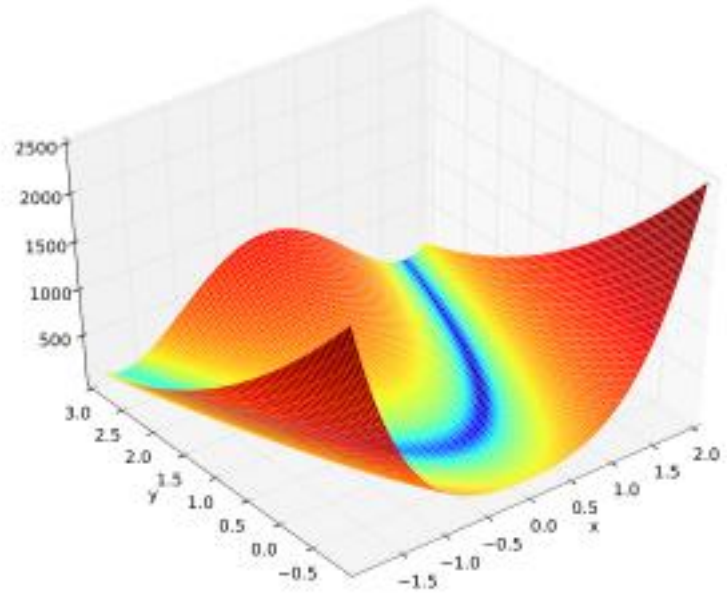
- Potential topics:
  - Circuit simulator that outputs time/voltage traces in columnar tabular data; want to extract voltage 4 at time 0, time final, and its max.
  - CASL VERA, which outputs on reactor geometry to binary HDF5; want to extract the outlet pressure and the average pin power.
  - Finite element analysis: force integrated over a sideset or nodeset.
- Tools: shell commands, Python, Perl, Matlab, C, C++, Fortran, Java, VBS, ParaView, VTK, VisIt, etc.

Interfacing

EXAMPLE AND EXERCISE

Example: Rosenbrock Black Box

Rosenbrock Function



Courtesy Wikipedia
https://en.wikipedia.org/wiki/Rosenbrock_function

- See discussion in Section 10.3 of the Dakota User's Manual

Scenario: Coat Hook Design

Scenario: Your manager would like to place some coat hooks in your building's lobby. To ensure they will safely hold heavy winter coats, he asked you to conduct a computational study, suggesting they are similar to cantilever beams.

You immediately assign this critical task to a trusted intern. She downloads the latest version (12.1.4) of *Cantilever Physics*, an advanced cantilever beam simulation tool developed by Sandia National Laboratories, and gets to work. After she shows you the results of a few test cases, you remain uncertain about the design of the coat hooks. It occurs to you that Dakota could help you to achieve greater confidence in your analysis, and you ask her to begin developing an interface between Dakota and Cantilever Physics.
Unfortunately, the summer ends before she is able to finish, leaving you on the hook to complete the job.

Your task: Complete the Dakota-Cantilever Physics interface that your intern left unfinished.

Exercise: Cantilever Beam Interface

Exercise materials are located in `~/exercises/interfacing`. They include:

- The Cantilever Physics executable, `cantilever`, along with an old input file named `cantilever.i`. Try running it:

```
./cantilever cantilever.i
```

- A Dakota input file, `dakota_cantilever.in`
- An example Dakota parameters file, `params.in`, which you can use for testing. (Generate your own `params.in` by adding the `file_save` keyword to the Dakota input file.)
- `driver.sh`, a partially-complete interface script

General Interfacing Advice

- Think about automation from the beginning
- Sketch the workflow you are automating, and how Dakota will interact with it
- Test each component/step (dakota input, pre-processing, run, post-processing) in isolation
- Capture the results of each step to log files
- If a step fails, how would I know?
  - Silent failures, where an erroneous value is returned to Dakota rather than a crash, are the most dangerous!
- Avoid long chains of tools that multiply points of failure

- Treat your interface as though your conclusions depend on it!

Interfacing

DAKOTA INPUT

Dakota Input - Interface Block

The interface section of the input file tells Dakota how to run a simulation

Example:

```
interface
 analysis_drivers = 'driver.sh'

  fork

  parameters_file = 'params.in'
  results_file = 'results.out'

  file_tag
  file_save
```

Interface executable name

Interface type

Names of param. and results files

Save and tag param. and results file

Discussion: Browse to the Dakota Reference Manual, interface section and call out some keywords that look particularly helpful.

Excursis: Fork vs System vs Direct

- **`direct`** interface is used to run an analysis driver that has been linked or compiled into Dakota
  - Largely beyond the scope of this training
  - One matter of practical interest: text_book
  - See the Developer's Manual on the website for more information
- **`fork`** and **`system`** are used to launch an external simulation
  - The fork interface waits for your analysis driver to exit, then tries to open your results file
  - The system interface polls the OS for the existence of the results file, and tries to read it as soon as it is written
  - Practical consequence: When using the system interface, don't create the results file until the very end of your interface script
  - *In general, we recommend the fork interface*

What else can Dakota do?

- Work directories separate evaluations

```
work_directory
  named  'workdir'
  directory_save directory_tag
```

- Naming optional
- Directories deleted by default
- Dakota writes params file to each work directory, and launches interface from there.

- Work directories can share common template files

```
work_directory
 link_files  'template/input.template'
 copy_files  'modified/*'
```

- Copy or link files or entire directories into your work dir before launching interface
- Wildcards permitted

What else can Dakota do?

- Asynchronous execution helps with local parallelism

```
asynchronous
  evaluation_concurrency 2
```

- Run multiple evaluations concurrently
- By default, ALL available evals will run
- MUST use `file_tag` or `directory_tag`

Your Simulation

- Ideally, you will leave training this week with a working interface to a simulation of relevance to you
- Give it a shot this evening and discuss during office hours